Une généralisation de l'approche Cyclic-Clustering pour la résolution de CSP structurés

Cedric Pinto, Cyril Terrioux

Une généralisation de l'approche Cyclic-Clustering pour la résolution de CSP structurés

Cédric Pinto **Cyril Terrioux**

LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{cedric.pinto, cyril.terrioux}@lsis.org

Résumé

Nous proposons une nouvelle méthode pour résoudre des CSP structurés, qui généralise et améliore l'approche Cyclic-Clustering [6]. D'abord, l'ensemble coupe-cycle et la décomposition arborescente du réseau de contraintes, qui sont employés pour tirer profit de la structure du CSP, sont calculés indépendamment de la notion quelque peu contraignante de sous-graphe triangulé induit. Ensuite, contrairement à Cyclic-Clustering, la méthode proposée permet de résoudre le CSP induit par la décomposition arborescente sans avoir nécessairement instancié au préalable toutes les variables de la partie coupe-cycle. La résolution d'un tel CSP peut être effectuée grâce à la méthode BTD [8] comme dans [9]. Dans la mesure où BTD mémorise et exploite des (no)goods structurels, nous nous intéressons ensuite aux conditions qui rendent possible la réutilisation de (no)goods structurels enregistrés lors d'appels précédents à BTD. Ces conditions sont alors exploitées dans une version dédiée de BTD de sorte à profiter autant que possible des informations précédemment mémorisées.

Du point de vue théorique, la méthode proposée permet de garantir une borne de complexité en temps relative à des paramètres liés à l'ensemble coupe-cycle et à la décomposition arborescente employés. Concernant l'intérêt pratique, nous pouvons espérer détecter plus tôt les échecs et éviter certaines redondances dans la recherche. Cet intérêt pratique est évalué lors d'expériences préliminaires.

Abstract

We propose a new method for solving structured CSPs which generalizes and improves the Cyclic-Clustering approach [6]. First, the cutset and the tree-decomposition of the constraint network, which are used for taking advantage of the CSP structure, are computed independently of the notion of triangulated induced subgraph. Then, unlike Cyclic-Clustering, our method can try to solve the tree-decomposition part of the problem without having assigned all the variables of the cutset. Regarding the solving of the tree-decomposition part, we use the BTD method [8] like in [9]. As BTD records and exploits structural (no)goods, we provide some conditions which make possible the use of structural (no)goods recorded during previous calls of BTD and we implement them in a dedicated version of BTD. By so doing, from a theoretical viewpoint, we can provide a theoretical time complexity bound related to parameters of the cutset and the tree-decomposition and, from a practical viewpoint we expect to detect failures earlier and to avoid more redundancies in the search. This practical interest is assessed in some preliminary experiments.

1 Préliminaires

Le formalisme des problèmes de satisfaction de contraintes (CSP) offre un cadre de formalisation extrêmement puissant pour l'expression et la résolution d'une multitude de problèmes, en particulier de nombreux problèmes académiques ou issus du monde réel (par exemple les problèmes de coloration de graphes, d'ordonnancement, d'affectation de fréquence, ...). Un *problème de satisfaction de contraintes* (X, D, C, R) est défini comme un ensemble de variables $X = \{x_1, \ldots x_n\}$ soumis à un ensemble de contraintes $C = \{c_1, \ldots, c_m\}$. A chaque variable x_i est

associé un domaine d_i issu de l'ensemble de domaines $D = \{d_1, \dots d_n\}$. Le domaine fini d_i contient chacune des valeurs possibles pour x_i. Une contrainte $c_i \in C$ est un sous-ensemble ordonné de variables, $c_i = (x_{i_1}, \dots x_{i_{a_i}})$ (le nombre a_i de variables impliquées dans la contrainte c_i est appelé l'arité de cette contrainte). Notons que nous utilisons la même notation pour désigner la contrainte c_i et l'ensemble des variables sur lesquelles elle porte. A chaque contrainte c_i est associée une relation $r_{c_i} \in R$ définissant les combinaisons de valeurs autorisées pour les variables soumises à la contrainte c_i ($r_{c_i} \subseteq d_{i_1} \times \dots \times d_{i_{a_i}}$). Soit $Y = \{x_1, \dots x_k\}$ un sous-ensemble de X. Une *affectation* $\mathcal{A}$ sur Y est un tuple $(v_1, \dots, v_k)$ de $d_1 \times \dots \times d_k$. Nous écrirons également $\mathcal{A}$ sous la forme plus explicite $\{x_1 \leftarrow v_1, ..., x_i \leftarrow v_k\}$. Ensuite, nous noterons $\mathcal{A}_1 \subseteq \mathcal{A}_2$ si l'affectation A_2 est une extension de l'affectation A_1 (i.e. nous avons $\mathcal{A}_1 = \{x_1 \leftarrow v_1, ..., x_i \leftarrow v_i\}$ et $\mathcal{A}_2 = \{x_1 \leftarrow v_1, ..., x_i \leftarrow v_i, ..., x_{i+j} \leftarrow v_{i+j}\}$ avec $j \geq 0$). Une affectation $\mathcal{A}$ sur Y satisfait une contrainte $c \in C$ telle que $c \subseteq Y$ si $\mathcal{A}[c] \in r_c$ où $\mathcal{A}[c]$ désigne la restriction de $\mathcal{A}$ aux variables sur lesquelles porte c. $\mathcal{A}$ est dite *consistante* si elle satisfait chaque contrainte c telle que $c \subseteq Y$. Une solution est une affectation de chacune des variables qui satisfait toutes les contraintes. Déterminer s'il existe une solution constitue un problème NP-Complet. Nous noterons $Sol(\mathcal{P})$ l'ensemble des solutions du CSP $\mathcal{P}$. Pour des raisons de simplicité, nous ne considérerons, dans la suite de cet article, que le cas des CSP binaires (c'est-à-dire des CSP dont chaque contrainte porte sur deux variables exactement). Néanmoins, ces travaux peuvent s'étendre aux CSP n-aires.

Les méthodes utilisées habituellement pour résoudre des CSP (comme, par exemple, les méthodes Forward Checking [5] et MAC [11]) reposent généralement sur une recherche énumérative de type backtracking. Cette approche, souvent efficace en pratique, a une complexité en temps exponentielle en $O(m.d^n)$ (noté $O(exp(n))$) pour une instance ayant n variables et m contraintes et dont le plus grand domaine possède d valeurs. Plusieurs travaux ont été développés afin d'améliorer cette complexité théorique en exploitant des caractéristiques particulières de l'instance. Généralement, ils reposent sur certaines propriétés structurelles du CSP. La structure d'un CSP (X, D, C, R) peut être représentée par le graphe (X, C), appelé le *graphe de contraintes*. Dans ce contexte, la notion de décomposition arborescente [10] joue un rôle clé. Une *décomposition arborescente* d'un graphe $G = (X, C)$ est un couple(E, T) où $T = (I, F)$ est un arbre dont I est l'ensemble des nœuds et F celui des arêtes et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X telle que chaque sous-ensemble (appelé cluster) E_i est un nœud de T et vérifie : (i) $\cup_{i \in I} E_i = X$, (ii) pour chaque arête $\{x, y\} \in C$, il existe $i \in I$ avec $\{x, y\} \subseteq E_i$, et (iii) pour tout $i, j, k \in I$, si k est sur une chaîne entre i et j dans T, alors $E_i \cap E_j \subseteq E_k$. La largeur w d'une décomposition arborescente (E, T) est égale à $max_{i \in I}|E_i| - 1$. La *largeur d'arbre* (ou *tree-width*) w^* de G est la largeur minimale sur toutes les décompositions arborescentes de G. D'une part, cette notion conduit à une des meilleures bornes connues de complexité théorique en temps, à savoir $O(exp(w^* + 1))$ avec w^* la largeur d'arbre. Plusieurs méthodes (par exemple [4, 8]) ont été proposées pour atteindre cette borne. Elles visent à regrouper les variables entre elles (en formant des clusters) de sorte que l'arrangement en clusters forme un arbre. D'autre part, les décompositions arborescentes sont également employées dans d'autres méthodes structurelles, comme la méthode Cyclic-Clustering [6]. Plus précisément, la méthode Cyclic-Clustering consiste à instancier un sous-ensemble de variables (appelé ensemble coupe-cycle) tel que le graphe de contraintes du problème réduit aux variables non instanciées est un arbre de cliques (qui correspond donc à une décomposition arborescente).

D'un point de vue théorique, atteindre la meilleure borne de complexité théorique en temps requiert de calculer une décomposition arborescente optimale (c'est-à-dire une décomposition arborescente ayant une largeur minimale), ce qui constitue un problème NP-Difficile [1]. En pratique, il semble difficilement concevable de résoudre un problème NP-Difficile comme étape préliminaire de la résolution d'un problème NP-Complet. Aussi, au lieu d'utiliser une méthode exacte, on fait plutôt appel, en général, à des méthodes heuristiques. Souvent, ces méthodes fournissent de bonnes (voire de très bonnes) approximations d'une décomposition arborescente optimale quand le graphe de contraintes possède une petite largeur d'arbre. Des méthodes comme BTD [8] sont alors tout indiquées pour résoudre de tels problèmes. En revanche, quand la structure du graphe de contraintes est moins "jolie" (et donc que la largeur d'arbre est plus importante), ces méthodes heuristiques produisent trop souvent des décompositions arborescentes avec une largeur importante et conduisent ainsi à des approximations très grossières d'une décomposition arborescente optimale. Dans un tel cas, exploiter une méthode de résolution comme Cyclic-Clustering peut se révéler plus intéressant et surtout plus adapté. Cyclic-Clustering repose sur un sous-ensemble V de sommets (et donc de variables), appelé coupe-cycle, du graphe (X, C), tel que le sous-graphe $(X - V, \{\{x, y\} \in C \; tel \; que \; x, y \in X - V\})$ induit par $X - V$ soit triangulé (c'est-à-dire qu'il ne possède pas de cycle de longueur supérieure ou égale à 4 sans une arête joignant deux sommets non consécutifs dans le cycle). La partie triangulée du graphe de contraintes correspond en fait à une décomposition arborescente. A titre d'exemple, la figure 1(a) présente un graphe ayant 19 sommets. L'ensemble $\{y_1, y_2\}$ forme un coupe-cycle de ce graphe puisque le sous-graphe induit par $\{x_1, \dots, x_{17}\}$ est triangulé. Dans [9], deux implémentations de Cyclic-Clustering, appelées respectivement CC-

BTD_1 et CC-BTD_2, ont été proposées. Toutes deux résolvent d'abord la partie du problème associée à l'ensemble coupe-cycle avec un algorithme énumératif classique (comme FC par exemple), puis la partie associée au sous-graphe triangulé induit en employant la méthode BTD. CC-BTD_2 se distingue de CC-BTD_1 par un appel préliminaire à BTD avant d'entamer la résolution de la partie coupe-cycle. En procédant ainsi, les nogoods mémorisés lors de cet appel préliminaire à BTD restent valides durant l'ensemble de la résolution et donc dans les différents appels ultérieurs à BTD. L'approche Cyclic-Clustering (et plus particulièrement ses deux implémentations) possède plusieurs limites. D'une part, les informations mémorisées, sous forme de goods et de nogoods structurels durant les différents appels à BTD, ne sont pas exploitées ultérieurement (hormis les nogoods enregistrés durant l'appel préliminaire à BTD pour CC-BTD_2), conduisant ainsi à visiter plusieurs fois le même espace de recherche. D'autre part, la partie triangulée doit être calculée en exploitant la notion de sous-graphe triangulé induit (TIS).

Dans cet article, nous proposons une généralisation de CC-BTD, appelée CC-BTD-gen. A l'image de l'approche Cyclic-Clustering, CC-BTD-gen se base sur un ensemble coupe-cycle et une décomposition arborescente du graphe de contraintes. Toutefois, la décomposition arborescente utilisée peut être calculée avec n'importe quelle méthode, que cette méthode exploite ou non la notion de TIS. S'affranchir de la notion de TIS permet, par exemple, de pouvoir calculer un coupe-cycle puis d'en déduire une décomposition arborescente pour les variables qui ne participent pas au coupe-cycle. Concernant la résolution, CC-BTD-gen exploite une version dédiée de BTD qui permet de tirer profit d'une partie des (no)goods mémorisés lors d'appels antérieurs à BTD, permettant ainsi d'éviter, en pratique, un certain nombre de redondances dans la recherche. Enfin, nous avons constaté que CC-BTD instancie de façon consistante toutes les variables de la partie coupe-cycle avant de résoudre la partie associée à la décomposition arborescente, même si après avoir instancié quelques variables du coupe-cycle, il n'est plus possible de trouver une solution sur la partie triangulée. Par conséquent, afin d'éviter un tel inconvénient, CC-BTD-gen a la possibilité d'appeler BTD après avoir instancié de façon consistante seulement une partie des variables du coupe-cycle. Ainsi, si le sous-problème associé à la décomposition arborescente possède une solution, la recherche peut continuer sur les variables non instanciées du coupe-cycle. Sinon, on peut remettre en cause immédiatement l'affectation courante sur le coupe-cycle. Dans les deux cas, des goods et des nogoods pourront être enregistrés et réutilisés plus tard.

Ce papier est organisé ainsi. La section 2 présente le cadre formel de CC-BTD-gen tandis que la section 3 décrit l'algorithme lui-même. Ensuite, nous évaluons expérimentalement l'intérêt de cette approche dans la section 4. Enfin, nous concluons et discutons des travaux connexes ou à venir dans la section 5.

2 Cadre formel

Dans cette section, nous allons décrire le cadre formel requis pour pouvoir présenter proprement CC-BTD-gen. Ce cadre est ici défini dans un contexte général avant d'être utilisé ensuite dans le contexte spécifique de CC-BTD-gen où Y désignera l'ensemble coupe-cycle et $X - Y$ l'ensemble des variables impliquées dans la décomposition arborescente. Dans la suite, nous considérons un CSP $\mathcal{P} = (X, D, C, R)$. Dans un premier temps, nous définissons la notion de sous-problème induit par un sous-ensemble Y de variables.

Définition 1 *Soit $Y \subseteq X$ un sous-ensemble de variables. Le **CSP induit par** Y est le CSP (Y, D_Y, C_Y, R_Y) où $D_Y = \{d_i \in D | x_i \in Y\}$, $C_Y = \{c_{ij} = \{x_i, x_j\} \in C | x_i, x_j \in Y\}$ et $R_Y = \{r_{ij} \in R | c_{ij} \in C_Y\}$.*

Dans les définitions et propriétés suivantes, nous considérerons les notations suivantes :

- Y_1, Y_2, Y et Z seront des sous-ensembles de X tels que $Y_1 \subseteq Y$, $Y_2 \subseteq Y$, $Y \subseteq X$ et $Z \subseteq X - Y$,
- $\mathcal{A}_1$ une affectation sur Y_1, A_2 sur Y_2 et $\mathcal{A}$ sur Y,
- TD la décomposition arborescente utilisée pour le CSP $\mathcal{P}(X - Y)$,
- $S_j = E_i \cap E_j$ un séparateur de TD avec E_i et E_j deux clusters de TD tels que E_j soit un fils de E_i,
- $Desc(E_j)$ l'ensemble des variables appartenant à la descendance du cluster E_i enracinée en E_j.

Nous proposons maintenant une définition certes limitée, mais suffisante, de l'effet du filtrage effectué par l'algorithme Forward-Checking (FC [5]).

Définition 2 *Le filtrage résultant de l'affectation $\mathcal{A}$ accompli par FC est l'opération qui consiste à supprimer du domaine d_i de chaque variable non instanciée x_i les valeurs qui deviennent incompatibles vis-à-vis d'au moins une contrainte $\{x_i, y\}$ avec y une variable instanciée dans $\mathcal{A}$. Plus formellement, $d_i^{\mathcal{A}} = \{v \in d_i | \forall c = \{x_i, y\} \in C,\ (v, w) \in r_c\ avec\ w\ la\ valeur\ affectée\ à\ y\ dans\ \mathcal{A}\}$.*

En d'autres mots, $d_i^{\mathcal{A}}$ désigne le domaine courant d'une variable non instanciée x_i obtenu par l'application du filtrage de FC après chaque affectation d'une variable de $\mathcal{A}$. Avec le même objectif de caractériser l'effet du filtrage, nous définissons également l'ensemble des valeurs supprimées par le filtrage.

Définition 3 *Soient un sous-ensemble $Y \subseteq X$ tel que $|Y| = k$ et $\mathcal{A} = \{x_1 \leftarrow v_1, ..., x_k \leftarrow v_k\}$ une affectation sur Y. L'ensemble des valeurs supprimées de $\mathcal{P}(X - Y)$ par le filtrage résultant de l'affectation $\mathcal{A}$ est $\mathcal{F}_{\mathcal{A}}(X - Y) = \{(x_i, v) \in (X - Y) \times (d_i - d_i^{\mathcal{A}})\}$.*

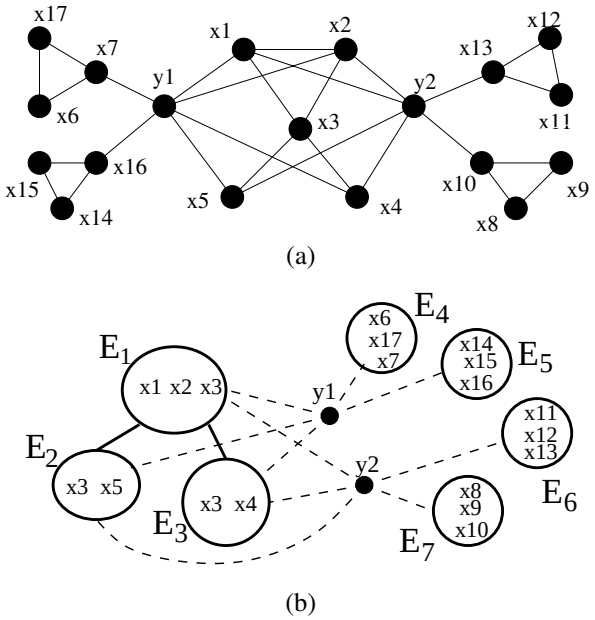

FIG. 1 – (a) Un graphe de contraintes, (b) un exemple de décomposition arborescente avec les clusters $E_1, \ldots, E_7$ et de coupe-cycle $\{y_1, y_2\}$ pour ce graphe.

Nous raffinons ensuite la définition 1 en introduisant la notion de problème filtré.

Définition 4 *Le problème filtré $\mathcal{P}_{\mathcal{A}}(X-Y)$ correspond au CSP $(X-Y, D^{\mathcal{A}}_{X-Y}, C_{X-Y}, R^{\mathcal{A}}_{X-Y})$ avec $D^{\mathcal{A}}_{X-Y} = \{d_i^{\mathcal{A}} | x_i \in X-Y\}$ et $R^{\mathcal{A}}_{X-Y} = \{r_c^{\mathcal{A}} = r_c \cap (d_j^{\mathcal{A}} \times d_k^{\mathcal{A}}) | c = \{x_j, x_k\} \in C_{X-Y}$ et $r_c \in R\}$.*

Nous pouvons constater que le filtrage effectué par FC ne modifie en rien la structure définie par le graphe de contraintes d'un problème. Il en est de même pour les décompositions arborescentes :

Propriété 1 *Une décomposition arborescente de $\mathcal{P}(X-Y)$ est une décomposition arborescente de $\mathcal{P}_{\mathcal{A}_1}(X-Y)$, et réciproquement.*

Preuve : Une décomposition arborescente ne dépend que du graphe considéré. Dans la mesure où la structure de $\mathcal{P}(X-Y)$ et celle de $\mathcal{P}_{\mathcal{A}_1}(X-Y)$ sont représentées par le même graphe de contraintes, ces deux problèmes possèdent nécessairement les mêmes décompositions arborescentes. □

A présent, nous mesurons l'impact du filtrage sur les domaines et les relations d'un problème donné au travers de la propriété suivante.

Propriété 2 *Si $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$, alors $\forall z_i \in Z$, $d_i^{\mathcal{A}_2} \subseteq d_i^{\mathcal{A}_1}$ et $\forall c_{jk} \in C_Z$, $r_{c_{jk}}^{\mathcal{A}_2} \subseteq r_{c_{jk}}^{\mathcal{A}_1}$.*

Preuve : Pour chaque couple $(z_i, v_i) \in \mathcal{F}_{A_1}(Z)$, le filtrage consiste à supprimer la valeur v_i du domaine de la variable z_i. Donc $\forall d_i^{\mathcal{A}_1} \in D^{\mathcal{A}_1}_{X-Y}$, $d_i^{\mathcal{A}_1} = d_i - \{v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{\mathcal{A}_1}(Z)\}$. De même, nous avons $\forall d_i^{\mathcal{A}_2} \in D^{\mathcal{A}_2}_{X-Y}$, $d_i^{\mathcal{A}_2} = d_i - \{v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{\mathcal{A}_2}(Z)\}$. Toutefois, comme $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$, nous avons $\{v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{\mathcal{A}_1}(Z)\} \subseteq \{v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{\mathcal{A}_2}(Z)\}$. Il en résulte que $d_i^{\mathcal{A}_2} \subseteq d_i^{\mathcal{A}_1}$.

Soient $c_{jk} \in C_Z$ une contrainte entre deux variables $x_j, x_k \in Z$ et $r_{c_{jk}}^{\mathcal{A}_1}$ et $r_{c_{jk}}^{\mathcal{A}_2}$ les relations associées obtenues consécutivement au filtrage résultant respectivement de $\mathcal{A}_1$ et de $\mathcal{A}_2$. D'après la définition 4, nous avons $r_{c_{jk}}^{\mathcal{A}_1} = r_{c_{jk}} \cap (d_j^{\mathcal{A}_1} \times d_k^{\mathcal{A}_1})$ et $r_{c_{jk}}^{\mathcal{A}_2} = r_{c_{jk}} \cap (d_j^{\mathcal{A}_2} \times d_k^{\mathcal{A}_2})$. De plus, $d_j^{\mathcal{A}_2} \times d_k^{\mathcal{A}_2} \subseteq d_j^{\mathcal{A}_1} \times d_k^{\mathcal{A}_1}$ puisque $\forall z_i \in Z$, $d_i^{\mathcal{A}_2} \subseteq d_i^{\mathcal{A}_1}$. Donc, $r_{c_{jk}}^{\mathcal{A}_2} \subseteq r_{c_{jk}}^{\mathcal{A}_1}$. □

Nous comparons maintenant les ensembles de solutions de deux sous-problèmes induits par le même sous-ensemble de variables mais résultant de filtrages différents.

Propriété 3 *Si $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$, alors $Sol(\mathcal{P}_{\mathcal{A}_2}(Z)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_1}(Z))$ et $|Sol(\mathcal{P}_{\mathcal{A}_2}(Z))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Z))|$.*

Preuve : Soit $\mathcal{S}$ une solution de $\mathcal{P}_{\mathcal{A}_2}(Z)$. Montrons que $\mathcal{S}$ est aussi une solution de $\mathcal{P}_{\mathcal{A}_1}(Z)$. Par définition, $\mathcal{S}$ est une affectation consistante de toutes les variables de Z telle que $\forall c_{jk} \in C_Z$, $\mathcal{S}[c_{jk}] \in r_{c_{jk}}^{\mathcal{A}_2}$. Or, $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$ et, d'après la propriété 2, $\forall c_{jk} \in C_Z$, $r_{c_{jk}}^{\mathcal{A}_2} \subseteq r_{c_{jk}}^{\mathcal{A}_1}$. Par conséquent, $\forall c_{jk} \in C_Z$, $\mathcal{S}[c_{jk}] \in r_{c_{jk}}^{\mathcal{A}_1}$. Donc, $\mathcal{S}$ est bien une solution de $\mathcal{P}_{\mathcal{A}_1}(Z)$. D'où $Sol(\mathcal{P}_{\mathcal{A}_2}(Z)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_1}(Z))$ et $|Sol(\mathcal{P}_{\mathcal{A}_2}(Z))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Z))|$. □

Le corollaire suivant précise ce qu'il en est dans le cas particulier d'une affectation $\mathcal{A}_2$ extension d'une affectation $\mathcal{A}_1$.

Corollaire 1 *Si $\mathcal{A}_2[Y_1] = \mathcal{A}_1$, alors $Sol(\mathcal{P}_{\mathcal{A}_2}(Z)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_1}(Z))$ et $|Sol(\mathcal{P}_{\mathcal{A}_2}(Z))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Z))|$.*

Preuve : Il suffit d'observer que l'hypothèse $\mathcal{A}_2[Y_1] = \mathcal{A}_1$ implique que $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$. Par conséquent, en exploitant la propriété 3, on obtient $Sol(\mathcal{P}_{\mathcal{A}_2}(Z)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_1}(Z))$ et $|Sol(\mathcal{P}_{\mathcal{A}_2}(Z))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Z))|$. □

Nous rappelons, dans la définition suivante, la notion de goods et de nogoods structurels [8].

Définition 5 *Etant donnés deux clusters E_i et E_j avec E_j un fils de E_i, un **good** (respectivement un **nogood**) structurel de E_i par rapport à E_j est une affectation consistante $\mathcal{A}$ sur $S_j = E_i \cap E_j$ telle que $\mathcal{A}$ peut (resp. ne peut pas) être étendue en une affectation consistante sur $Desc(E_j)$.*

Nous pouvons désormais caractériser les cas pour lesquels les (no)goods structurels d'un sous-problème $\mathcal{P}(X-Y)$ restent valides si nous changeons d'affectation sur Y.

Théorème 1 *Si $\mathcal{F}_{\mathcal{A}_1}(X-Y) \subseteq \mathcal{F}_{\mathcal{A}_2}(X-Y)$ et $ng(S_j)$ est un nogood du sous-problème $\mathcal{P}_{\mathcal{A}_1}(X-Y)$, alors $ng(S_j)$ est aussi un nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}(X-Y)$.*

Preuve : TD est une décomposition arborescente associée aux CSP $\mathcal{P}_{\mathcal{A}_1}$ et $\mathcal{P}_{\mathcal{A}_2}$. De plus, $S_j = E_i \cap E_j$ est un séparateur de TD entre le cluster E_i et un de ses fils E_j. Nous savons que $ng(S_j)$ est un nogood pour $\mathcal{P}_{\mathcal{A}_1}(X-Y)$. Donc $|Sol(\mathcal{P}_{\mathcal{A}_1}(Desc(E_j)))| = 0$ si l'affectation sur S_j correspond à ce nogood. Cependant, $\mathcal{F}_{\mathcal{A}_1}(X-Y) \subseteq \mathcal{F}_{\mathcal{A}_2}(X-Y)$ et $Desc(E_j) \subseteq X-Y$. Par conséquent, nous pouvons appliquer la propriété 3. Il en découle que $|Sol(\mathcal{P}_{\mathcal{A}_2}(Desc(E_j)))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Desc(E_j)))|$ et donc que $|Sol(\mathcal{P}_{\mathcal{A}_2}(Desc(E_j)))| = 0$. Autrement dit, $ng(S_j)$ est aussi un nogood pour $\mathcal{P}_{\mathcal{A}_2}(X-Y)$. □

Le théorème précédent pose une condition (inclusion) sur l'ensemble des valeurs supprimées par le filtrage pour pouvoir conclure à la validité ou non d'un nogood déjà mémorisé. Cependant, d'un point de vue algorithmique et pratique, exploiter ce théorème semble conduire à un test trop coûteux à la fois en temps et en espace. Aussi, dans le corollaire ci-dessous, nous proposons une restriction sur les filtrages résultant de deux affectations.

Corollaire 2 *Si $\mathcal{A}_2[Y_1] = A_1$ et $ng(S_j)$ est un nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}_1}(X-Y)$, alors $ng(S_j)$ est aussi un nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}(X-Y)$.*

Preuve : En observant que $\mathcal{A}_2[Y_1] = \mathcal{A}_1$ implique que $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$, il suffit d'appliquer le théorème 1 pour obtenir le résultat souhaité. □

Ensuite, nous nous intéressons à conserver la validité des goods.

Théorème 2 *Si $\mathcal{F}_{\mathcal{A}_2}(Desc(E_j)) \subseteq \mathcal{F}_{\mathcal{A}_1}(Desc(E_j))$ et $g(S_j)$ est un good pour le sous-problème $\mathcal{P}_{\mathcal{A}_1}(X-Y)$, alors $g(S_j)$ est aussi un good pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}(X-Y)$.*

Preuve : Comme $\mathcal{F}_{\mathcal{A}_2}(Desc(E_j)) \subseteq \mathcal{F}_{\mathcal{A}_1}(Desc(E_j))$ et $Desc(E_j) \subseteq X-Y$, nous pouvons appliquer la propriété 3. Par conséquent, $Sol(\mathcal{P}_{\mathcal{A}_1}(Desc(E_j)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_2}(Desc(E_j)))$. De plus, nous savons que $g(S_j)$ est un good pour le sous-problème $\mathcal{P}_{\mathcal{A}_1}(X-Y)$. Donc, $\mathcal{P}_{\mathcal{A}_1}(Desc(E_j))$ possède au moins une solution $\mathcal{S}$ (par définition d'un good). Donc, $\mathcal{S}$ est une solution du sous-problème $\mathcal{P}_{\mathcal{A}_2}(Desc(E_j))$ également. Par conséquent, $g(S_j)$ est un good pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}(X-Y)$. □

Toutes ces propriétés et corollaires peuvent être utilisés dans le cadre de l'algorithme CC-BTD-gen afin de décider quelles sont les informations qui demeurent valides entre les différents appels à BTD. Pour cela, Y correspondra à l'ensemble coupe-cycle et donc $X-Y$ à l'ensemble des variables appartenant à la décomposition arborescente associée. Le théorème 1 permet de conclure qu'à partir de deux affectations partielles $\mathcal{A}_1$ et $\mathcal{A}_2$ sur le coupe-cycle telles que $\mathcal{A}_2$ filtre au moins les mêmes valeurs que $\mathcal{A}_1$, les nogoods mémorisées par BTD pour le sous-problème $\mathcal{P}_{\mathcal{A}_1}$ restent valides pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}$. Cependant, du fait de la quantité d'espace mémoire limitée, nous ne pouvons pas nous permettre de mémoriser précisément les effets du filtrage découlant de chaque affectation partielle consistante sur le coupe-cycle. Aussi, nous exploiterons plutôt le corollaire 2, qui rend possibles l'enregistrement et la réutilisation de nogoods dans le cas où nous étendons une affectation partielle consistante sur le coupe-cycle. De même, pour la réutilisation des goods, nous nous contenterons de tester la validité des goods au moment de les utiliser afin de limiter le coût global en temps de ces tests. Dans la section suivante, nous décrivons et étudions l'algorithme CC-BTD-gen.

3 Une généralisation de Cyclic-Clustering

L'algorithme CC-BTD-gen (voir algorithme 1) repose sur l'exploitation d'un ensemble coupe-cycle et d'une décomposition arborescente du graphe de contraintes. Ce coupe-cycle et cette décomposition arborescente peuvent être calculés grâce à n'importe quelle méthode, cette dernière pouvant utiliser ou non la notion de sous-graphe triangulé induit (TIS). L'algorithme CC-BTD-gen consiste à instancier de façon consistante les variables du coupe-cycle tout en vérifiant, grâce à une version dédiée de BTD, si l'affectation courante sur le coupe-cycle peut être étendue de manière consistante sur la partie correspondant à la décomposition arborescente. Ce test pouvant s'avérer coûteux, après chaque affectation consistante d'une variable du coupe-cycle, la méthode CC-BTD-gen décide, par l'intermédiaire de la fonction heuristique $ChoiceBTD$, si elle doit ou non l'effectuer. Si BTD retourne $true$, la méthode CC-BTD-gen continue la recherche sur le coupe-cycle. Dans le cas contraire, elle essaie une nouvelle valeur (s'il en reste) pour la variable courante du coupe-cycle ou revient en arrière sur la variable précédente. Ce processus est itéré jusqu'à l'obtention d'une solution (c'est-à-dire d'une affectation consistante du coupe-cycle qui peut être étendue de manière consistante sur la partie correspondant à la décomposition arborescente) ou jusqu'à ce que l'ensemble de l'espace de recherche ait été exploré.

En premier lieu, afin de pouvoir réutiliser les (no)goods enregistrés lors des différents appels à BTD, nous propo-

Algorithme 1 : CC-BTD-gen($in : \mathcal{A}, V, NG_p, in/out : G_p$)

```
1  Cons ← true
2  if ChoiceBTD(V) or V = ∅ then
3      G ← ∅ ; NG ← ∅
4      Cons ← BTD-gen(∅, E_1, V_{E_1}, NG_p, G_p, NG, G)
5      G_p ← G_p ∪ G ; NG_p ← NG_p ∪ NG
6  if Cons and V ≠ ∅ then
7      Choose x_i ∈ V ; d_i ← D_i ; Cons ← false
8      while d_i ≠ ∅ and ¬Cons do
9          Choose v ∈ d_i ; d_i ← d_i − {v}
10         if Filtering(A ∪ {x_i ← v}, x_i) then
11             Cons ←
               CC-BTD-gen(A ∪ {x_i ← v}, V − {x_i}, NG_p, G_p)
12         Unfiltering(A, x_i)
13 return Cons
```

Algorithme 2 : BTD-gen($in : \mathcal{A}, E_i, V_{E_i}, NG_p, G_p, in/out : NG, G$)

```
1  if V_{E_i} = ∅ then
2      Cons ← true ; F ← Sons(E_i)
3      while F ≠ ∅ and Cons do
4          Choose E_j ∈ F ; F ← F − {E_j}
5          S_j ← E_i ∩ E_j ;
6          if A[S_j] is a nogood into NG then Cons ← false
7          else if A[S_j] is a nogood into NG_p then Cons ← false
8          else if A[S_j] is a good into G then Cons ← true
9          else if A[S_j] is a good into G_p and CheckGood(E_j, A[S_j])
           then Cons ← true
10         else
11             Cons ←
               BTD-gen(A, E_j, E_j \(E_j ∩ E_i), NG_p, G_p, NG, G)
12             if Cons then Save the good A[S_j] into G
13             else Save the nogood A[S_j] into NG
14 else
15     Choose x_k ∈ V_{E_i} ; d_k ← D_k ; Cons ← false
16     while d_k ≠ ∅ et ¬Cons do
17         Choose w ∈ d_k ; d_k ← d_k − {w}
18         if A ∪ {x_k ← w} satisfies each constraint then
19             Cons ←BTD-gen(A ∪ {x_k ← w}, E_i, V_{E_i} − {x_k},
               NG_p, G_p, NG, G)
20 return Cons
```

Algorithme 3 : CheckGood(E_i, g)

```
1  Let S be the assignment g and its recorded extension on E_i
2  forall y ∈ E_i do
3      if S[y] ∉ d_y then return false
4  ValidGood ← true ; F ← Sons(E_i)
5  while F ≠ ∅ et ValidGood do
6      Choose E_j ∈ F ; F ← F − {E_j}
7      g_F ← good on E_j such that g_F[E_i ∩ E_j] = S[E_i ∩ E_j]
8      ValidGood ←CheckGood(E_j, g_F)
9  return ValidGood
```

sons une variante de BTD, appelée BTD-gen (voir algorithme 2), qui met en œuvre les propriétés mises en évidence dans la section précédente. BTD-gen ne se distingue de BTD que par sa capacité à exploiter des (no)goods mémorisés durant des appels antérieurs à BTD-gen. Par conséquent, cette variante est dotée de deux paramètres supplémentaires, à savoir l'ensemble G_p des goods et l'ensemble NG_p des nogoods qui ont été enregistrés lors des exécutions précédentes de BTD-gen. Quant aux ensembles G et NG, ils désignent respectivement l'ensemble des goods et celui des nogoods produits durant l'appel courant à BTD-gen. Dans la mesure où nous conservons l'intégralité des goods produits, certains d'entre eux peuvent ne pas demeurer valides dans certains appels à BTD-gen. Par conséquent, avant de réutiliser un tel good, BTD-gen doit préalablement s'assurer de sa validité vis-à-vis du sous-problème courant afin de respecter le théorème 2. Ce test est accompli par le biais de la fonction $CheckGood$ (voir algorithme 3). Cette fonction renvoie $true$ si chacune des variables de la descendance de E_i peut être instanciée avec la valeur qu'elle possédait au moment où le good g a été mémorisé. Afin de pouvoir tester facilement cette propriété, nous avons besoin de mémoriser l'extension du good sur l'ensemble des autres variables du cluster. Comme BTD, BTD-gen retourne la consistance du sous-problème associé à la décomposition arborescente TD et enraciné en E_i.

Cette version dédiée de BTD est exploitée dans CC-BTD-gen pour vérifier que l'affectation partielle courante sur le coupe-cycle peut être étendue de façon consistante sur la partie correspondant à la décomposition arborescente. Si BTD-gen($\emptyset, E_1, V_{E_1}, NG_p, G_p, NG, G$) renvoie $false$, alors CC-BTD-gen tente d'instancier la variable courante du coupe-cycle avec une nouvelle valeur (s'il en reste). En l'absence d'autres valeurs, CC-BTD-gen revient en arrière sur la variable précédente. Dans le cas où l'appel à BTD-gen renvoie $true$, CC-BTD-gen continue la recherche en instanciant une nouvelle variable du coupe-cycle. Dans les deux cas, l'ensemble G des goods découverts lors de l'appel à BTD-gen est ajouté à l'ensemble G_p. Le processus est similaire pour les nogoods, si ce n'est que NG_p ne peut être modifié au delà de l'appel courant à CC-BTD-gen. Autrement dit, à chaque retour en arrière d'un appel à CC-BTD-gen, nous oublions les nogoods mémorisés durant cet appel, afin de respecter le corollaire 2.

Enfin, la fonction booléenne $ChoiceBTD$ définit, après chaque affectation d'une variable du coupe-cycle, si BTD-gen doit être appelé ou non. Si elle retourne $false$ et qu'il reste des variables non instanciées dans le coupe-cycle, CC-BTD-gen va essayer d'en affecter une avec l'algorithme Forward-Checking (lignes 7-12). Si $ChoiceBTD$ renvoie $true$, CC-BTD-gen fait appel à BTD-gen et conserve les nouveaux goods et nogoods mémorisés (lignes 3-5). Notons que cette heuristique peut être entièrement dynamique. Elle peut donc décider d'appeler BTD-gen après n'importe quelle affectation partielle consistante du coupe-cycle.

Nous illustrons maintenant l'algorithme CC-BTD-gen avec un exemple. Considérons, pour cela, le graphe de contraintes de la figure 1(a) et une décomposition arborescente avec cinq composantes connexes et un coupe-cycle contenant deux variables y_1 et y_2 comme représenté dans la figure 1(b). Supposons que CC-BTD-gen commence par instancier des variables du coupe-cycle. Par exemple, s'il

a affecté seulement la variable y_1, le filtrage de FC qui s'ensuit a pu réduire le domaine des variables non instanciées voisines de y_1, à savoir x_1, x_2, x_4, x_5, x_7 et x_{16}. Ensuite, l'heuristique $ChoiceBTD$ peut décider de résoudre le sous-problème associé à la décomposition arborescente avec BTD-gen. Si BTD-gen retourne $false$, alors CC-BTD-gen va modifier l'affectation de y_1. Sinon, l'algorithme CC-BTD-gen va tenter d'instancier de manière consistante la variable y_2 du coupe-cycle. S'il y parvient, un nouvel appel à BTD-gen est accompli puisque toutes les variables du coupe-cycle sont instanciées. Si BTD-gen renvoie $true$, alors le CSP est consistant. Dans le cas contraire, CC-BTD-gen va essayer une nouvelle valeur pour y_2, et s'il n'en reste plus va revenir sur y_1.

Théorème 3 *BTD-gen est correct, complet et termine.*

Preuve : BTD est correct, complet et termine [8]. Comme BTD-gen ne se distingue de BTD que par l'exploitation des (no)goods enregistrés durant les appels précédents à BTD-gen, nous devons simplement prouver que l'utilisation de ces (no)goods ne remet pas en cause la validité, la complétude et la terminaison de BTD. Si Y désigne le coupe-cycle, $X-Y$ correspond aux variables de la décomposition arborescente. D'après le corollaire 2, si $\mathcal{A}$ est une affectation consistante sur Y, alors chaque nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}}(X-Y)$ est également un nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}'}(X-Y)$ avec $\mathcal{A}'$ une extension consistante de $\mathcal{A}$ sur le coupe-cycle. De plus, lorsqu'on revient en arrière de $\mathcal{A}' = \mathcal{A} \cup \{x_k \leftarrow w\}$ vers $\mathcal{A}$, CC-BTD-gen oublie tous les nogoods qui ont été mémorisés depuis l'affectation de la variable x_k à la valeur w. Par conséquent, le corollaire 2 est bien respecté et il est bien valide d'utiliser les nogoods de NG_p. Concernant l'exploitation des goods de G_p, si la fonction $CheckGood$ renvoie $true$ pour un good donné, il s'ensuit qu'utiliser ce good est valide d'après le théorème 2. Aussi, comme l'algorithme BTD-gen n'emploie que des (no)goods valides, il est correct, complet et termine. □

Théorème 4 *CC-BTD-gen est correct, complet et termine.*

Preuve : Afin de prouver plus aisément la correction, la complétude et la terminaison de CC-BTD-gen, nous allons considérer l'algorithme avec un point de vue légèrement différent. La résolution effectuée par CC-BTD-gen peut être décomposée en deux phases. La première phase revient à résoudre la partie coupe-cycle avec une version modifiée de l'algorithme FC. Cette dernière consiste simplement à utiliser l'algorithme FC classique et, quand l'heuristique $ChoiceBTD$ renvoie $true$, à appeler BTD-gen. Cet appel à BTD-gen peut être vu comme un test de consistance supplémentaire. En effet, on vérifie si l'affectation partielle courante sur le coupe-cycle peut être étendue ou non de manière consistante sur les variables de la décomposition arborescente. La validité de cette coupe supplémentaire ne dépend bien sûr que de la validité de BTD-gen.

La seconde phase débute quand toutes les variables du coupe-cycle sont instanciées de façon consistante. Cette affectation constitue donc une solution du sous-problème lié au coupe-cycle, qui doit être étendue de manière consistante sur la partie associée à la décomposition arborescente. Cette seconde phase est accomplie à l'aide de BTD-gen.

Dans la mesure où les algorithmes FC et BTD-gen sont corrects, complets et terminent, il en est nécessairement de même pour l'algorithme CC-BTD-gen. □

Dans les deux théorèmes suivant, nous noterons n le nombre de variables du CSP, m le nombre de contraintes, d la taille du plus grand domaine, k la taille du coupe-cycle, w la largeur de la décomposition arborescente considérée, et s la taille de la plus grande intersection entre deux clusters de la décomposition arborescente.

Théorème 5 *BTD-gen a une complexité en temps en $O(n(n+m)d^{w+1})$ et une complexité en espace en $O(nwd^s)$.*

Preuve : La preuve est similaire à celle de BTD [8]. Nous devons juste prendre en compte le coût en temps additionnel nécessaire pour tester la validité des goods de G_p et l'espace supplémentaire requis pour mémoriser l'extension de chaque good sur le cluster correspondant. □

Théorème 6 *CC-BTD-gen a une complexité en temps en $O(n(n+m)d^{w+k+2})$ et une complexité en espace en $O(nwd^s)$.*

Preuve : Dans le pire des cas, CC-BTD-gen appelle BTD-gen après chaque affectation d'une variable du coupe-cycle. Comme le nombre d'affectations partielles du coupe-cycle est borné par d^{k+1}, CC-BTD-gen a une complexité en temps en $O(n(n+m)d^{w+1}.d^{k+1} + nm.d^{k+1})$, c'est-à-dire en $O(n(n+m)d^{w+k+2})$. Pour la complexité en espace, cela dépend uniquement de BTD-gen. Donc, CC-BTD-gen a une complexité en espace en $O(nwd^s)$. □

4 Résultats expérimentaux

Dans cette section, nous évaluons expérimentalement l'intérêt pratique de l'algorithme CC-BTD-gen par rapport à des méthodes structurelles classiques à savoir CC-BTD$_1$, CC-BTD$_2$ et BTD et à une méthode énumérative classique, en l'occurrence FC. Les tests sont effectués sur des problèmes structurés générés aléatoirement. Plus précisément, nous utilisons un générateur de CSP binaires. Ce générateur construit d'abord un premier sous-problème dont le graphe de contraintes est triangulé. Puis, il génère un second sous-problème qui correspond à l'ensemble coupe-cycle. Enfin, il relie les deux sous-problèmes en ajoutant

Classes $(\mathbf{n}, \mathbf{d}, \mathbf{r}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3}, \mathbf{s}, \mathbf{k}, \mathbf{e_1}, \mathbf{e_2})$	BTD						CC-BTD (1,2,gen)		
	min-fill		Fusion						
			$\mathbf{CC_{gen}}$		BY		BY		
	w	s	w	s	w	s	k	w	s
(a) (120, 15, 15, 65, 70, 40, 5, 15, 80, 30)	40,7	7	40,8	9,2	54,5	9,1	13,9	13,9	4,9
(b) (120, 15, 15, 65, 80, 30, 5, 15, 80, 30)	40,7	7	27,2	14,4	38,3	14,1	13,9	13,9	4,9
(c) (150, 15, 15, 65, 70, 40, 5, 15, 65, 30)	54	6	42,3	9,3	59	9,5	14,2	13,9	5
(d) (150, 15, 15, 65, 80, 20, 5, 15, 50, 30)	38,6	7	42	9,2	56,2	9,7	13,3	14	5
(e) (150, 15, 15, 64, 60, 60, 5, 15, 50, 30)	30	8	42	9,2	56,2	9,7	13,3	14	5
(f) (200, 15, 15, 64, 30, 30, 5, 15, 30, 20)	36	6	34,2	9,3	42,1	9,7	12	13,9	5

TAB. 1 – Paramètres des différentes classes et valeurs des paramètres structurels des différentes méthodes considérées. Le coupe-cycle et la décomposition arborescente associée sont calculés à partir de l'algorithme de Balas et Yu (BY) ou sont ceux produits par le générateur aléatoire (CC_{gen}).

Classes	FC		BTD				CC-BTD						CC-BTD-gen					
			min-fill		Fusion		$CC\text{-}BTD_1$		$CC\text{-}BTD_2$		H_k		H2		H1	
(a)	$>_8$	281	$>_{23}$	585	$>_{10}$	240	$>_{27}$	649	$>_1$	25.3	$>_1$	25		3,51		1,80
(b)	$>_8$	264	$>_{19}$	489		4,99		24,6		0,84		1,05		5,49		6,44
(c)	$>_8$	260	$>_{26}$	643	$>_{20}$	625	$>_{32}$	773	$>_3$	76	$>_3$	74,8		14,28		0,70
(d)	$>_5$	195	$>_{41}$	993	$>_{33}$	839	$>_{31}$	748	$>_2$	52,4	$>_2$	51,9		4,13		0,34
(e)	$>_{10}$	317	$>_{41}$	986	$>_{41}$	986	$>_{33}$	795	$>_5$	124	$>_5$	121	$>_3$	82,1		15,91
(f)	$>_{15}$	605	$>_{42}$	1008	$>_{39}$	953	$>_{34}$	816	$>_6$	144	$>_6$	144	$>_5$	120	$>_1$	24,4

TAB. 2 – Temps d'exécution (en secondes) pour les différentes méthodes sur les classes considérées. Le coupe-cycle et la décomposition arborescente associée sont ceux produits par le générateur aléatoire (CC_{gen}).

aléatoirement un certain nombre de contraintes. Dix paramètres sont nécessaires pour générer de tels problèmes, à savoir n, d, r, t_1, t_2, t_3, s, k, e_1 et e_2. L'instance générée comporte $n + k$ variables qui ont toutes un domaine de taille d. Plus précisément, la partie triangulée possède n variables réparties dans des cliques de taille au plus r et dont l'intersection entre deux cliques est de taille au plus s. La partie coupe-cycle est composée de k variables et e_1 contraintes. Enfin, e_2 contraintes relient ces deux parties entre elles. Les relations associées à chaque contrainte possèdent t_1 tuples interdits pour les contraintes entre deux variables de la partie triangulée, t_2 pour les contraintes entre deux variables du coupe-cycle et t_3 pour les contraintes liant une variable du coupe-cycle à une variable de la partie triangulée. Par conséquent, une classe de ces problèmes aléatoires est définie par la donnée de ces dix paramètres : $(n, d, r, t_1, t_2, t_3, s, k, e_1, e_2)$. Pour chaque classe considérée, le nombre de problèmes consistants est approximativement le même que celui de problèmes inconsistants. Dans nos expérimentations, BTD et BTD-gen exploitent l'algorithme FC pour résoudre chaque cluster. Concernant l'ordre d'instanciation des variables dans FC ou au sein d'un cluster pour BTD et BTD-gen, nous utilisons l'heuristique *dom/deg* qui choisit comme prochaine variable la variable x_i qui minimise le rapport $\frac{|d_{x_i}|}{|\Gamma_{x_i}|}$ avec d_{x_i} le domaine courant de de x_i et Γ_{x_i} l'ensemble des variables voisines de x_i.

Nous comparons les résultats obtenus par CC-BTD-gen avec ceux de BTD, $CC\text{-}BTD_1$ et $CC\text{-}BTD_2$ d'une part, et de FC d'autre part. Pour des raisons d'efficacité, les méthodes basées sur l'approche Cyclic-Clustering classique, comme $CC\text{-}BTD_i$, nécessitent un ensemble coupe-cycle ayant peu de solutions. Malheureusement, à notre connaissance, aucune méthode satisfaisante n'existe pour déterminer une telle structure. Nous utiliserons donc d'une part l'algorithme de Balas et Yu [2] qui permet de calculer un sous-graphe induit triangulé à partir duquel est ensuite déduit le coupe-cycle. D'autre part, nous emploierons également le coupe-cycle et la décomposition arborescente utilisés lors de la génération de l'instance. L'idée en procédant ainsi est simplement d'observer le comportement de ces algorithmes quand une structure a priori adéquate est exploitée.

Pour CC-BTD-gen, et plus particulièrement, pour la fonction $ChoiceBTD$, nous avons testé plusieurs heuristiques, notées H_i avec $i = \{1, ..., k\}$. Chaque heuristique H_i décide de résoudre la partie associée à la décomposition arborescente si, depuis le dernier appel à BTD-gen, au moins i variables du coupe-cycle ont été instanciées de façon consistante et si au moins une valeur a été supprimée par filtrage dans un domaine associé à une variable de la décomposition arborescente. Initialement, chaque heuristique effectue un appel préliminaire à BTD-gen, à l'image de l'appel initial à BTD accompli par $CC\text{-}BTD_2$. Dans ce papier, nous ne présenterons que les résultats pour les heuristiques H_1, H_2 et H_k.

Classes	BTD		CC-BTD				CC-BTD-gen					
	Fusion		CC-BTD$_1$		CC-BTD$_2$		H$_k$		H2		H1	
(a)	$>_{21}$	574	$>_{32}$	792	$>_6$	168	$>_6$	157	$>_3$	107	$>_1$	28,2
(b)	$>_9$	297	$>_{22}$	617	$>_7$	186	$>_5$	155	$>_1$	41,3	$>_1$	35,5
(c)	$>_{28}$	712	$>_{41}$	1016	$>_{14}$	339	$>_{13}$	313	$>_{12}$	289	$>_6$	197
(d)	$>_{31}$	771	$>_{36}$	877	$>_7$	181	$>_7$	177	$>_4$	106	$>_4$	96,3
(e)	$>_{36}$	881	$>_{33}$	795	$>_7$	171	$>_7$	170	$>_4$	125	$>_3$	77,4
(f)	$>_{40}$	981	$>_{39}$	936	$>_{13}$	312	$>_{13}$	312	$>_{11}$	264	$>_{10}$	240

TAB. 3 – Temps d'exécution (en secondes) pour les différentes méthodes sur les classes considérées. Le coupe-cycle et la décomposition arborescente associée sont calculés à partir de l'algorithme de Balas et Yu.

Les expérimentations sont effectuées sur un PC sous Linux doté d'un Pentium IV 3.2 GHz d'Intel et d'un 1 Go de mémoire vive. Les temps d'exécution sont exprimés en secondes. Pour chaque classe, nous lançons la résolution sur 50 instances. Les résultats présentés pour chaque classe sont des moyennes des résultats obtenus sur ces 50 instances. La notation $>_i$ indique que i instances n'ont pu être résolues par l'algorithme correspondant dans le temps imparti (à savoir 20 minutes). Dans un tel cas, comme le temps de résolution réel n'est pas connu, nous ajoutons une pénalité de 20 minutes par instance non résolue.

Dans un premier temps, nous pouvons observer une augmentation importante du temps d'exécution quand on passe de l'heuristique H_1 à l'heuristique H_k (excepté pour la seconde classe). En particulier, si H_1 et H_2 sont relativement proches conceptuellement, en pratique CC-BTD-gen a un comportement nettement plus performant avec H_1 qu'avec H_2. D'un point de vue théorique, dans le pire des cas, si $i < j$, alors H_i est susceptible d'appeler BTD-gen plus souvent que H_j. Cependant, en pratique, nous avons constaté que souvent, tester la consistance de la partie associée à la décomposition arborescente permet d'élaguer significativement l'espace de recherche relatif à la partie coupe-cycle du problème. Pour cette raison, H_1 obtient les meilleurs résultats par rapport aux autres heuristiques H_i, mais aussi aux méthodes CC-BTD$_1$ et CC-BTD$_2$. CC-BTD-gen avec H_k et CC-BTD$_2$ sont conceptuellement très proches. En effet, ces deux méthodes instancient intégralement le coupe-cycle avant de résoudre la partie correspondant à la décomposition arborescente. La seule différence notable est la réutilisation par CC-BTD-gen avec H_k des goods mémorisés lors des différents appels à BTD-gen. Cette différence se traduit généralement en pratique par des résultats meilleurs, ou dans le pire des cas comparables, pour CC-BTD-gen avec H_k par rapport à CC-BTD$_2$. Plus précisément, si, bien évidemment, les deux méthodes développent autant de nœuds sur la partie coupe-cycle, CC-BTD-gen en développe moins durant la résolution de la partie associée à la décomposition arborescente que CC-BTD$_2$, grâce aux goods mémorisés lors des appels précédents à BTD que CC-BTD-gen réutilise. Les cas où les deux méthodes sont équivalentes en temps correspondent alors souvent aux instances pour lesquelles l'apport des goods dans CC-BTD-gen n'est pas suffisant pour compenser le temps supplémentaire requis pour vérifier leur validité. Par ailleurs, comme cela a été mis en évidence dans [9], l'utilisation de nogoods mémorisés lors de l'appel préliminaire à BTD permet à CC-BTD$_2$ de résoudre plus d'instances que CC-BTD$_1$ et explique ainsi l'écart de temps important entre les deux méthodes. Concernant BTD, la quasi-totalité des instances n'ont pas une structure propice au calcul d'une décomposition arborescente par triangulation, ce qui explique les difficultés que rencontre BTD basé sur min-fill pour résoudre ces problèmes. C'est en particulier le cas, par exemple, quand le graphe associé à la partie coupe-cycle est peu dense et quand le coupe-cycle et la décomposition arborescente sont faiblement connectés. Quand la décomposition arborescente est calculée grâce à la méthode Fusion, BTD obtient des résultats meilleurs mais se révèle tout de même moins performant que CC-BTD-gen (excepté pour la classe (b)). Enfin, nous pouvons constater que FC rencontre également de grandes difficultés lors de la résolution de ces instances. En pratique, plusieurs instances de chaque classe ne sont pas résolues par FC et les temps de résolution sont globalement supérieurs à ceux obtenus par CC-BTD-gen.

Pour terminer, nous pouvons remarquer que CC-BTD$_i$ et CC-BTD-gen sont plus efficaces lorsqu'ils exploitent le coupe-cycle et la décomposition arborescente utilisés lors de la génération de chaque instance que lorsqu'ils emploient ceux produits via l'algorithme de Balas et Yu. Un tel résultat était prévisible. Il permet toutefois de souligner l'absence de méthodes pertinentes pour calculer à la fois un coupe-cycle et une décomposition arborescente de qualité vis-à-vis de la résolution, ce qui constitue un véritable problème pour de telles méthodes structurelles. Si l'algorithme de Balas et Yu permet effectivement de calculer un sous-graphe triangulé induit à partir duquel est déduit le coupe-cycle, il ne prend nullement en compte la résolution qui suivra. A titre d'exemple, pour les problèmes aléatoires que nous avons utilisés, cet algorithme obtient des valeurs de paramètres structurels (w, k et s) proches de celles utilisées pour la génération. Toutefois, en pratique, les coupe-cycle et décompositions arborescentes produites par cette

méthode conduisent à une résolution de moins bonne qualité. De plus, cet algorithme calcule souvent de grands ensembles coupe-cycle et des décompositions arborescentes triviales. Par exemple, nous l'avons testé sur des problèmes d'allocation de fréquence (à savoir les instances fapp de la dernière compétition de CSP [12]). A l'arrivée, la grande majorité des variables se trouve dans le coupe-cycle alors que la taille des clusters de la décomposition arborescente n'excède jamais 3. En plus de l'heuristique de Balas et Yu, nous avons essayé d'autres heuristiques pour calculer un coupe-cycle et une décomposition arborescente convenables (en utilisant ou non la notion de TIS). Malheureusement, à l'heure actuelle, aucune de ces heuristiques ne s'est révélée pertinente. Aussi, cela nous laisse à penser qu'une étude similaire à celle réalisée dans [7] pour les décompositions arborescente devra être menée afin d'améliorer l'efficacité pratique de ces méthodes structurelles. Dans le même temps, même si l'heuristique H_1 a fourni de bons résultats, il pourrait être intéressant de rechercher des heuristiques plus pertinentes pour la fonction $ChoiceBTD$.

5 Conclusion et discussion

Nous avons proposé une nouvelle méthode pour résoudre des CSP structurés. Cette méthode généralise et améliore l'approche Cyclic-Clustering [6]. Plus précisément, elle exploite un coupe-cycle et une décomposition arborescente dont le calcul est totalement indépendant de la notion de sous-graphe triangulé induit, ce qui apporte plus de liberté dans une phase cruciale de la méthode. Ensuite, CC-BTD-gen peut vérifier si l'affectation courante sur le coupe-cycle peut s'étendre de manière consistante sur la partie associée à la décomposition arborescente même si les variables de l'ensemble coupe-cycle ne sont pas toutes affectées. Cela permet alors à CC-BTD-gen d'avoir une vision plus globale du problème à résoudre. Enfin, CC-BTD-gen emploie une version dédiée de BTD qui met en œuvre des propriétés permettant d'exploiter des (no)goods mémorisés lors des précédents appels à BTD, évitant ainsi de nombreuses redondances dans la recherche. Les expérimentations préliminaires ont montré l'intérêt pratique de notre approche. En particulier, CC-BTD-gen s'avère significativement meilleur que CC-BTD$_i$.

Dans le cadre des CSP, peu de travaux ont été développés autour de la notion de coupe-cycle couplé avec une décomposition arborescente. [3] présente une méthode proche de Cyclic-Clustering dans laquelle la partie associée à la décomposition arborescente est résolue avec la méthode Adaptive-Consistency. De même, à notre connaissance, le calcul simultané de coupe-cycle et de décomposition arborescente pertinents en terme de résolution de CSP n'a pas encore été étudié à ce jour. Il va de soi qu'un tel travail devra être mené afin d'améliorer l'efficacité pratique de ce type d'approches, à l'image de l'étude réalisée dans [7] sur les décompositions arborescentes. En particulier, cela pourrait s'avérer très utile dans le cadre de la résolution d'instances issues du monde réel. Enfin, l'exploitation d'un coupe-cycle et d'une décomposition arborescente dynamiques pourrait se révéler fort prometteuse.

Références

[1] S. Arnborg, D. Corneil, and A. Proskuroswki. Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics*, 8 :277–284, 1987.

[2] E. Balas and C. Yu. Finding a maximum clique in an arbitrary graph. *Siam Journal on Computing*, 15(4) :1054–1068, 1986.

[3] R. Dechter. *Constraint processing*. Morgan Kaufmann Publishers, 2003.

[4] R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38 :353–366, 1989.

[5] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[6] P. Jégou. Cyclic-Clustering : a compromise between Tree-Clustering and the Cycle-Cutset method for improving search efficiency. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 369–371, 1990.

[7] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proc. of CP*, pages 777–781, 2005.

[8] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[9] P. Jégou and C. Terrioux. A Time-space Trade-off for Constraint Networks Decomposition. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 234–239, 2004.

[10] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[11] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 125–129, 1994.

[12] M. R. C. van Dongen, C. Lecoutre, and O. Roussel, editors. *Third International CSP Solver Competition*, 2008. `http://cpai.ucc.ie/08/`.